FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING
DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

MASTER'S THESIS

Formal connectivity verification of clock and reset signals in ultra-low-power SoC designs

Author	Aleksi Knuutinen
Supervisor	Jukka Lahti
Second Examiner	Jussi Jansson
(Technical Advisor	Juuso Rantanen)

December 2020

Knuutinen A. (2020) Formal connectivity verification of clock and reset signals in ultra-low-power SoC designs. University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 36 p.

ABSTRACT

This thesis investigates the usage of formal connectivity verification on clock and reset signal connectivity in ultra-low-power SoC designs. The origin of power consumption in CMOS circuits is explained, and the conflict between dynamic and static power on system parameter level is introduced. Common power reduction techniques are introduced and explained in some detail.

Overview of functional verification and its role in the design flow is presented. The main classification of functional verification into logic simulation and formal verification is discussed, and details of both are explained and compared. Challenges rising from low power design methodologies are introduced. Detailed view of connectivity and integration in SoC designs is provided, and a specified method of verifying connectivity is introduced in the form of formal connectivity verification.

The practical part of the thesis starts with an explanation of the verification goal and requirements for achieving it. Structure of the design environment used in the verification task is explained, and the different stages that the verification was conducted on. Creation of used connectivity properties and the used process flow for the chosen software tool is presented.

The process of confirming falsified properties as design bugs is introduced. The results of the verification task are presented, providing the total target amount for each verification stage, as well as the found bugs. The found bugs and their circumstances are explained. Comparison is made between the conventional method of verifying connectivity and the investigated formal method. Results show a great decrease in overall work effort, resourcing and time spent on the connectivity verification.

Key words: connectivity, formal verification, power management

Knuutinen A. (2020) Formaali liitettävyysverifiointi kello- ja reset-signaaleille ultra-matalan tehonkulutuksen järjestelmäpiireissä. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 36 p.

TIIVISTELMÄ

Tämä diplomityö tutkii formaalin liitettävyysverifionnin käyttöä kello- ja reset-signaalien yhteyksille ultra-matalan tehonkulutuksen järjestelmäpiireissä. Tehonkulutuksen lähteet CMOS piireissä selitetään, ja esitetään konflikti dynaamisen ja staattisen tehonkulutuksen välillä systeemin parametritasolla. Tavanomaisia tehonkulutusta vähentäviä tekniikoita esitellään ja selitetään jossain määrin.

Funktionaalisen verifioinnin yleiskatsaus ja asema suunnitteluvuossa esitellään. Funktionaalisen verifioinnin pääjaottelua logiikkasimulaatioon ja formaaliin verifiointiin käsitellään, ja molempien yksityiskohtia selitetään ja vertaillaan. Matalan tehonkulutuksen metodologioiden aiheuttamat ongelmat esitetään. Yksityiskohtainen kuvaus liitettävyydestä ja integroinnista järjestelmäpiireissä selitetään, ja eritelty metodi liitettävyyden verifioimiselle esitellään formaalin liitettävyysverifionnin muodossa.

Käytännön osuus diplomityöstä alkaa verifoinnin tavoitteen ja vaatimusten esittelemisellä. Käytetyn mallin rakenne ja verifiointitehtävä selitetään, sekä eri tasot joilla verifiointi suoritettiin. Liitettävyys-ominaisuuksien luominen, sekä käytetty prosessivuo valitulle työkalulle esitetään.

Vääriksi todistettujen ominaisuuksien varmistaminen suunnitteluvirheiksi esitellään. Tulokset verifointitehtävästä esitellään, käsitellen verifioinnin kohteiden kokonaista lukumäärää molemmilla verifiointitasoilla, sekä niistä löydettyjen virheiden määrää. Löydetyt suunnitteluvirheet ja niiden seikkaperät selitetään. Vertailua tehdään perinteisen liitettävyyden verifionnin metodin ja tutkitun formaalin metodin välillä. Tulokset osoittavat suuren säästön kokonaisessa työmäärässä, resurssoinnissa sekä liitettävyyden verifiointiin kulutetussa ajassa.

Avainsanat: liitettävyys, formaali verifiointi, virranhallinta.

TABLE OF CONTENTS

FOREWORD

The aim of this thesis was to investigate formal connectivity verification as a suitable method for verifying clock and reset signal connectivity in ultra-low-power SoC designs. The thesis work was conducted at Nordic Semiconductor Finland during the summer and autumn of 2020.

I would like to thank my manager Pekka Kotila for this great opportunity to work and conduct this thesis work at Nordic Semiconductor. Special thanks to Senior R&D Engineer Juuso Rantanen as the technical advisor and University Lecturer Jukka Lahti as the main supervisor for guidance and supervision during this thesis. Thanks to Jussi Jansson for being the second examiner. Finally, big thanks to all my colleagues who have always been ready to help and guide me when needed.

Oulu, December 9, 2020

Aleksi Knuutinen

LIST OF ABBREVIATIONS AND SYMBOLS

AMBA	Advanced Microcontroller Bus Architecture
CMOS	Complementary Metal Oxide Semiconductor
CPF	Common Power Format
CPU	Central Processing Unit
CSV	Comma-Separated Values
DSP	Digital Signal Processor
DUT	Design Under Test
DVFS	Dynamic Voltage and Frequency Scaling
EDA	Electronic Design Automation
FSM	Finite-State Machine
GPU	Graphical Processing Unit
HDL	High-level Description Language
IC	Integrated Circuit
I/O	Input/output
IP	Intellectual Property
RTL	Register Transfer Level
SoC	System-on-Chip
TCL	Tool Command Language
UPF	Unified Power Format
VHDL	VHSIC Hardware Description Language
WIFI	Wireless Fidelity
YAML	Yet Another Multicolumn Layout

C_{load}	Load capacitance
C_{ox}	Gate capacitance
I_{leak}	Leakage current
I_{SUB}	Sub-threshold leakage current
L	Length of a transistor
n	Variable of device fabrication
P_{leak}	Leakage power consumption
P_{sw}	Switching power consumption
V_{dd}	Supply voltage
V_{GS}	Gate-Source voltage
V_T	Threshold voltage
V_{th}	Thermal voltage
W	Width of a transistor

α	Switching activity probability
f	Frequency
μ	Carrier mobility

1 INTRODUCTION

The continued growth of complexity of system-on-chip (SoC) has forced the advance of power reduction technologies. As semiconductor development has steadily advanced according to Moore's Law [1], said advancement in performance has also increased power consumption in circuits. While transistor technology has enjoyed its rapid development, power storing technologies didn't have the same story and have advanced at a much slower rate. Unfortunately, "there is no Moore's Law for batteries" [2]. To combat rising power consumption, developers have resorted to the development of low-power SoC technologies and techniques. This thesis was conducted at Nordic Semiconductor, one of the leading developers of wireless SoC technology that focuses on the development and production of ultra-low-power SoC devices.

Power management systems in ultra-low-power SoC designs can be very complex, a result of parallel use of multiple different power reduction techniques at multiple design abstraction levels. This directly adds to the difficulty in SoC verification. Normally simple and straightforward signal connectivity, such as those of clock and reset signals, can become quite tedious to verify with typical simulation-based methods. Formal verification may offer the solution, with its specified "applications" for verifying design connectivity. [3]

This thesis explores the usage of formal connectivity verification on clock and reset signal connectivity in ultra-low-power SoC designs. The connectivity of these signals has become much more difficult to verify with typical simulation-based methods and should have more thorough verification. The primary goal is to investigate if formal connectivity verification method could work as a better and easier alternative to simulation-based methods in this kind of a verification task. As a direct result of the primary goal, the secondary goal is to explore the possibility of adding formal connectivity verification as a core part of the in-house verification flow.

Very little research can be found on this specific topic, as nearly all related research discusses formal connectivity verification and other forms of SoC connectivity verification in a very broad sense, not on any specific targets or verification tasks. This lack of specified research may be due to formal connectivity verification still being fairly new, or due to the research done being very product or company specific, and thus not being publicly available.

Chapters 2-4 focus on the theory concerning this work. Chapter 2 introduces the basic structure and features of SoC designs, as well as the basics of semiconductor power consumption. Chapter 3 provides an introduction of functional verification and presents the main methods of verification: simulation-based verification and formal verification. Chapter 4 delves into SoC connectivity and its possible types, and presents formal connectivity verification as a powerful method for verifying connectivity at IP, subsystem or the top SoC level.

Chapters 5-6 focus on the practical work in this thesis. Chapter 5 presents the used verification environment and targets, as well as the procedure for creating the connectivity properties to be checked. Chapter 6 presents the results of the task and compares the investigated method to the conventional verification method.

Chapter 7 discusses the aim of the thesis and the gained results. Chapter 8 is a recap of this thesis, where the main contents and results are shortly summarized.

2 ULTRA-LOW-POWER SYSTEM-ON-CHIP

Chapter 2.1 introduces the basic architecture of SoC designs. Chapter 2.2 introduces the basics of IC power consumption; how is it defined and classified, and how technology advancements have affected it. Chapter 2.3 provides few solutions to SoC power management.

2.1 SoC architecture

System-on-chip is an integrated circuit that integrates various computer components on a single chip. These components include a central processing unit (CPU), one or multiple memories, input/output ports, and some interface for communicating with the user or other devices. In addition to these, an SoC might be integrated with more advanced peripherals, such as Wireless Fidelity (WIFI) modules, digital signal processors (DSP), graphical processing units (GPU), or sensors for data collecting.

In SoC designs, these components are usually presented in forms of predesigned models of complex functions known as cores. These cores may be designed in-house or bought from separate core/Intellectual Property (IP) vendors. Cores are generally available in either synthesizable high-level description language (HDL) form, for example in Verilog or VHSIC Hardware Description Language (VHDL), or optimized transistor-level layout such as GDSII. [4]

Figure 1 demonstrates an example of a standard SoC design architecture, with multiple peripherals in the form of IPs, a memory block, one DSP core, and one CPU core. The components communicate through a data bus, which usually follows some form of Advanced Microcontroller Bus Architecture (AMBA) protocol. Input/output (I/O) interface exists for communication outside the chip.

A more advanced and complex SoC design may include multiple CPUs, memories for each CPU, and hundreds of IPs. In such a case, the top-level of the design may be divided into separate sections called subsystems. These subsystems are not dependent on other SoC components, and they contain all necessary components for a specific task, without overlapping with other functions on the chip [5]. A WIFI-subsystem for example handles everything related to WIFI communication and doesn’t need to rely on other cores like DSP or GPU for its functionality.

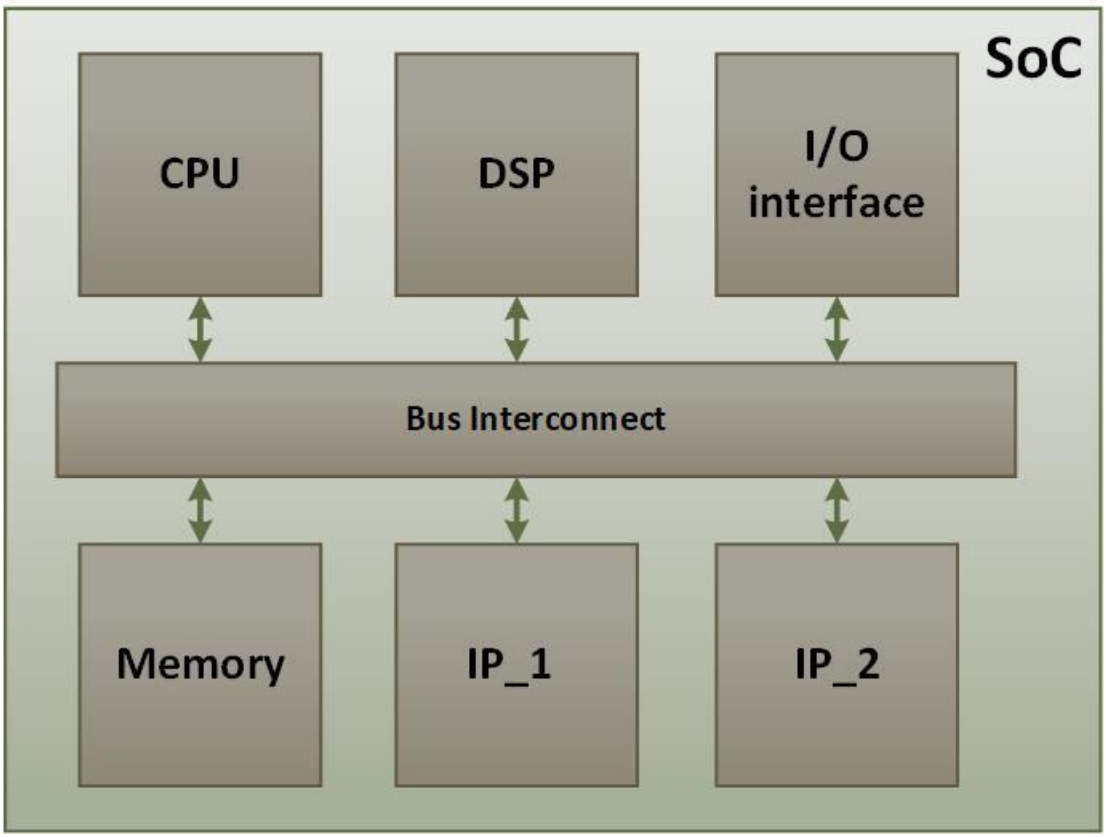

Figure 1: Example architecture of a standard SoC.

2.2 Semiconductor power consumption basics

As SoC chips get increasingly complex, they also consume increasing amounts of power. Maintaining a considerate power usage for a design is important, as high power consumption increases the cost of device usage and its manufacturing costs while decreasing its reliability and lifetime. This is especially important for portable battery-powered devices, which comprise one of the fastest-growing segments of the electronics market [11]. While Complementary Metal Oxide Semiconductor (CMOS) technology advances have seen transistor density doubling roughly every 18 months, the equivalent advancement for battery technology is greater than every 5 years [6].

Before attempting to reduce power consumption, one must first understand where it originates and what affects it. Typically, Integrated Circuit (IC) power consumption is divided into two parts, dynamic power and static power. In CMOS circuits, dynamic power is based on switching activity of CMOS logic gates, and static power is based on leakage currents flowing through transistors that are connected to a supply voltage. [7]

2.2.1 Dynamic power

Dynamic power is the power that is consumed when the device is active, meaning when signal values are changing. Dynamic power consumption is mostly caused by switching power consumption, which is the power dissipated by the charging and discharging of capacitance in the cell, and can be described as in the following Equation 1. [8]

$$P_{sw} = C_{load} \times V_{dd}^2 \times f \times \alpha, \quad (1)$$

where variables are as following:

- C_{load} is load capacitance
- V_{dd} is used supply voltage
- f is clock frequency
- α is switching activity probability.

As seen in the equation, reducing supply voltage V_{dd} is an effective way to reduce dynamic power. However, lowering the supply voltage also affects performance, as the speed of a gate decreases with a decrease in supply voltage [9]. A decrease in performance is very undesirable, and usually, it must be compensated in some way, so this approach needs to be carefully planned before execution.

2.2.2 Static power

Static power is the consumed power when the circuit isn't active, meaning when input and output signals aren't changing. Even if a circuit is turned off, as long as it is receiving power, it experiences power dissipation in the form of leakage power, which is power that does not contribute to the circuit's function. Leakage power dissipation occurs in both active and inactive states of the device, but the main concern with leakage power is during inactivity. Leakage power can be expressed as in the following Equation 2 [10].

$$P_{leak} = V_{dd} \times I_{leak} \quad (2)$$

where variables are as following:

- V_{dd} is used supply voltage
- I_{leak} is leakage current of transistor.

Looking at this equation, it seems as if it would be quite easy to reduce static power consumption by simply lowering supply voltage, as with reducing dynamic power. The problem however, is that in order to maintain high performance, a lower V_{dd} value needs to be compensated with a lower threshold voltage V_T. Changes in threshold voltage have an exponential effect on sub-threshold leakage current, which is one of the main components of total leakage current. The effect can be seen in the following approximation for I_{SUB} in Equation 3. [11]

$$I_{SUB} = \mu C_{ox} V_{th}^2 \times \frac{W}{L} \times e^{\frac{V_{GS} - V_T}{nV_{th}}}, \quad (3)$$

where variables are as following:

- μ is carrier mobility
- C_{ox} is gate capacitance
- V_{th} is the thermal voltage (25.9mV at room temperature)
- W is the width of transistor

- L is the length of transistor
- n is a variable of the device fabrication, varying from 1.0 to 2.5
- V_{GS} is the gate-source voltage
- V_T is the threshold voltage

As seen from the equation, a conflict is created. To reduce dynamic power, we lower V_{DD}, and to maintain performance, V_T is correspondingly lowered. This in turn exponentially increases leakage current, and thus increases leakage power.

Traditionally, managing leakage power has been redundant, but the advance of semiconductor technology has made it an increasing concern. The supply voltage of a transistor is lowered with each successive process generation, from 5V at 800nm technology size, to only 1.0V at 65nm technologies. As the technology size gets smaller and smaller, the resulting leakage power increases to a point where in some designs, the leakage power is nearly the same as dynamic power. Figure 2 shows how leakage power is catching up to dynamic power as the technology size scales further down[12].

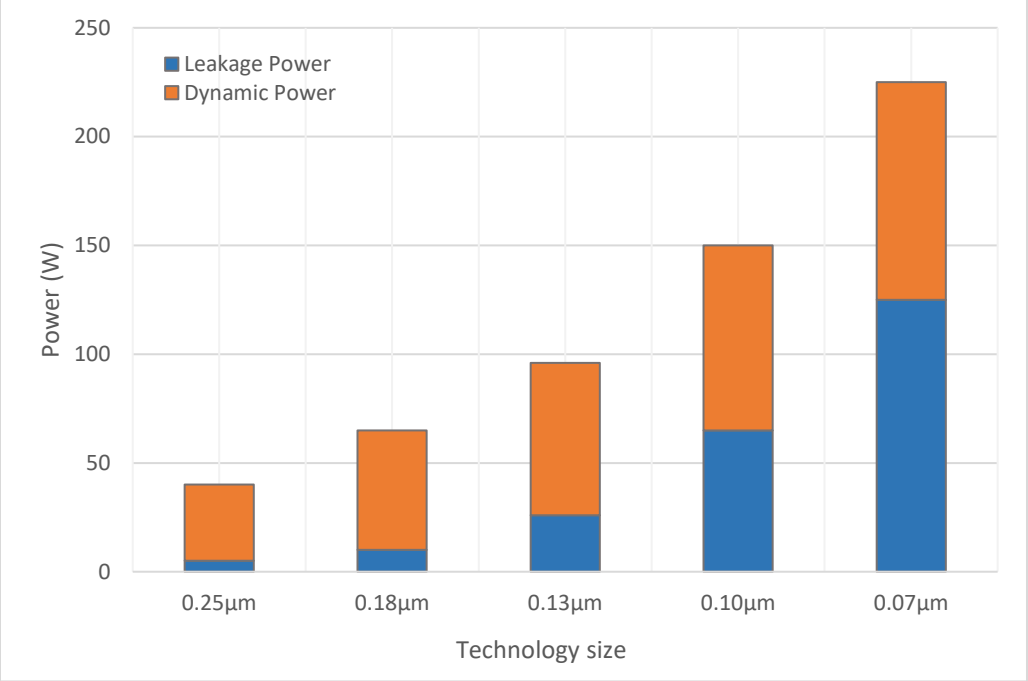

Figure 2: Leakage power vs. dynamic power in a CMOS chip at different technologies.

2.3 SoC power management solutions

Increasing power consumption is a huge problem for modern powerful electronics. Dynamic power consumption directly affects a device's operating time, the length of which is a more and more appreciated feature. Leakage power affects all devices that spend much of their operating time in some form of standby mode, such as cell phones. Lowering power consumption is highly desirable even for non-battery-powered devices, as it can reduce expenses in packaging and cooling. The recent rise of demand for ecological and sustainable production, and through it the demand for "greener" electronics, further drives the development of low power circuitry[13].

Design decisions made on the higher design abstraction levels can have a huge impact on the total power dissipation of a design, while design decisions on the lower levels of abstraction have a significantly smaller impact, as illustrated in Figure 3. Optimization on the lower levels might not be able to overcome optimization mistakes done on the higher levels in order to stay on the power budget of the design. To successfully reduce power consumption of the design as much as possible while avoiding design reiterations, optimization should be done continuously on all abstraction levels of the design. [14]

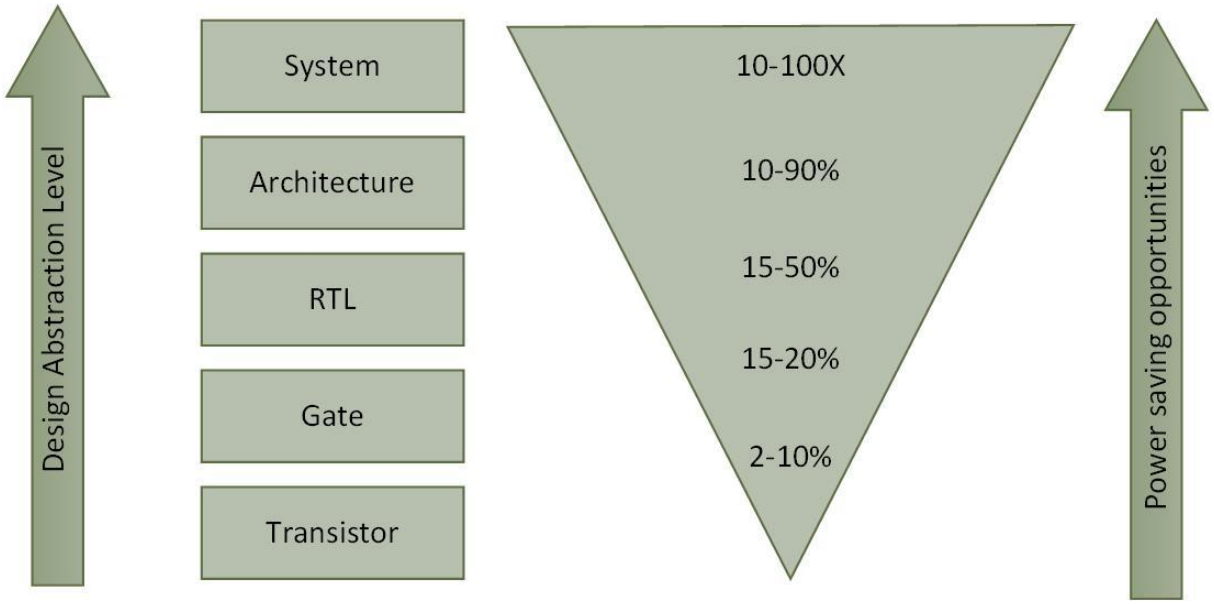

Figure 3. Power reduction opportunities in relation to design abstraction levels.

As power consumption in CMOS circuits can be divided into dynamic and static components, it is convenient to categorize power reduction techniques in the same way into dynamic and static categories. Many reduction methods span over multiple design levels, thus classification by design abstraction level can be difficult. Furthermore, this categorization reflects well to real life, where power reduction should be performed at every design abstraction level and design step. [14]

The following subheadings introduce and explain some of the commonly used power reduction techniques which were present and relevant in the practical part of this thesis. Some of the common but unmentioned power reduction techniques, such as Dynamic Voltage and Frequency Scaling (DVFS), were not relevant in the practical part, and thus are not needlessly touched upon.

2.3.1 Clock-gating

A large amount of dynamic power consumption in a design is in the clock network. Clock signals switch state on every cycle, so they naturally have the highest toggle rate in the system. A connected clock in an idle module not only adds to the clock loading but can also cause spurious activity in the logic. As the spurious activity under these conditions is quite random, activity may actually be maximized instead of minimized [13]. The intuitive way to reduce this power is to disconnect clocks from modules when they are not being used. This commonly used technique is known as clock gating.

Clock gating describes logic where the distribution of a clock signal is controlled by an enabling signal. The enable for a group of flip-flops is asserted only when those flip-flops have switching activity, thus preventing unneeded clock pulses and activity at idle parts of the design. Although the basic logic is the same for all clock gating modules, the implementation techniques differ.

Figure 4 describes a commonly used clock gating module, based on the usage of a latch. The latch is opaque when the clock signal is high, and it is driven on negative clock cycles. This ensures that the enable signal feeding the AND gate doesn't change during positive clock cycles, ensuring that there are no glitches on the output clock. [15]

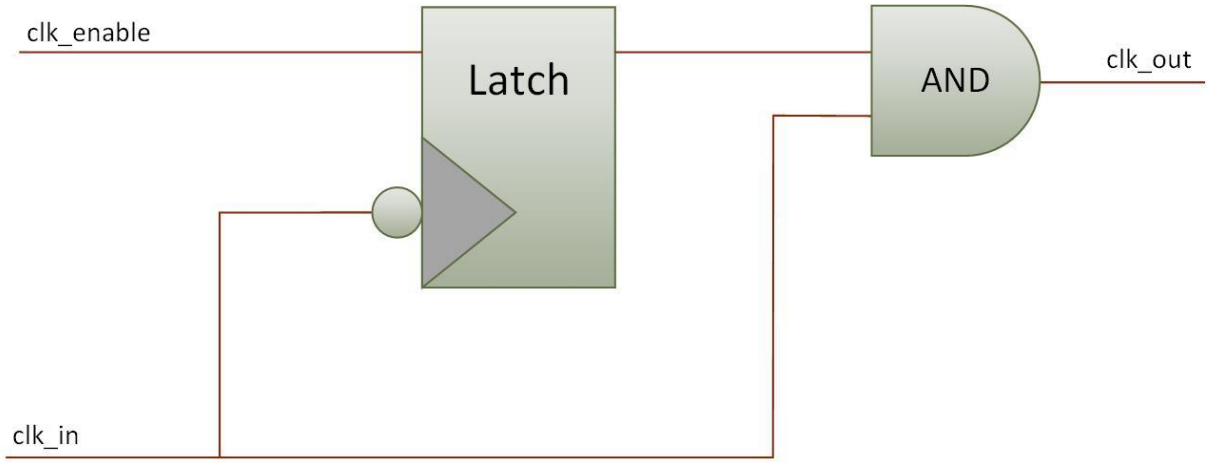

Figure 4: Latch based clock gating module.

2.3.2 Power-gating

As clock gating is a technique used for the reduction of dynamic power, power gating is a corresponding technique for static power reduction. Since leakage power dissipation always happens when circuits receive power, whether the circuits are active or not, the natural way to reduce it would be to turn off inactive modules by disconnecting their supply voltages. This can be done by using one PMOS and NMOS transistor in series with the module, with these transistors working as switches between the module, supply voltage, and ground. Depending on their position, these transistors are called "headers" or "footers". The usage of these transistor switches is depicted in Figure 5. Notice that in practice only one transistor is necessary. NMOS transistors are usually used because of their lower on-resistance, unless the circuit design has properties in favour of using PMOS header cells, such as multiple power supplies. [16]

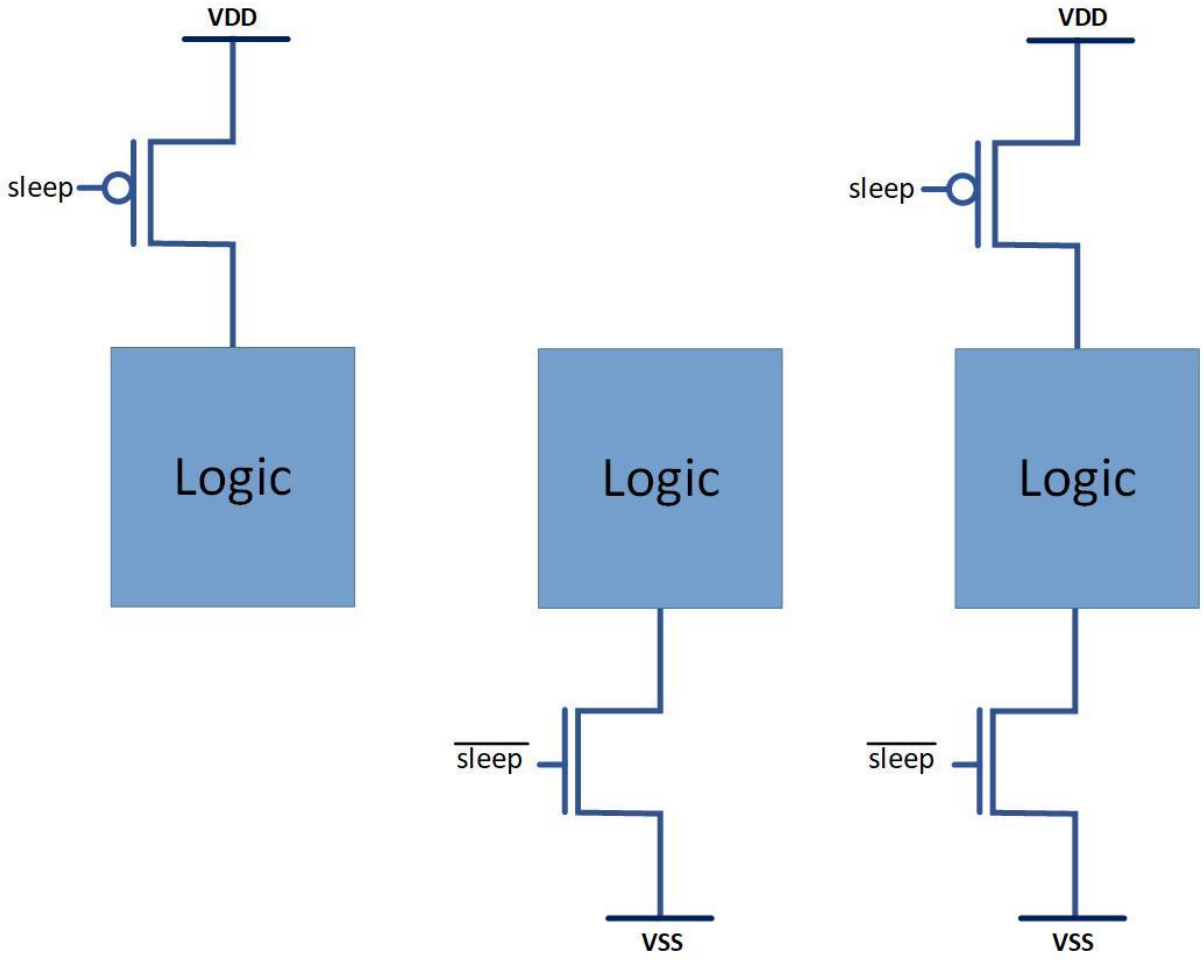

Figure 5. Power gating circuit structures.

During ACTIVE state, the transistors are on, and the circuit functions as usual. During SLEEP state, the transistors are turned off, disconnecting the path between supply power and ground, thus preventing leakage power dissipation from the logic. While the switching transistors are ON, the switching transistors themselves contribute to a small amount of leakage power consumption, but this is usually negligible in comparison to the leakage from the logic circuit.

2.3.3 Power domains

All sections of an SoC design are not equally active. Some modules may spend a lot of time in idle states and can be powered off temporarily with power gating to minimize leaking power consumption. The timing requirements can also differ between sections. In a standard single supply voltage circuit, the value of the supply voltage is determined by the clock requirement of the critical delay paths[17]. However, the number of critical paths typically constitutes only a small fraction of all paths within the design, and using the same supply voltage for every cell in the design therefore wastes energy[17].

Multi-supply voltage domain technique is a method, where the design is partitioned into separate voltage domains, each domain operating at a distinct power supply level depending on its timing requirements. Time-critical domains run at a higher supply voltage to maximize

performance, while non-critical sections work at lower supply voltage to reduce power consumption without impacting overall circuit performance. [18]

Figure 6 presents an example of an SoC design with 3 separate power domains. Power Domain 1 is an "always-on" domain, working with a lower frequency and thus requiring a lower supply voltage level. Power Domain 2 is reserved for time-critical modules, working at a higher clock frequency, thus also requiring a higher power supply. This high-speed domain is switched on only when required, due to its high power consumption. Power Domain 3 is a so-called "switchable" domain, which is switched on when deemed necessary. It serves as a domain with functionality somewhere between those of domains 1 and 2, not always being powered on, and not having the lowest or the highest clock frequency or power supply voltage.

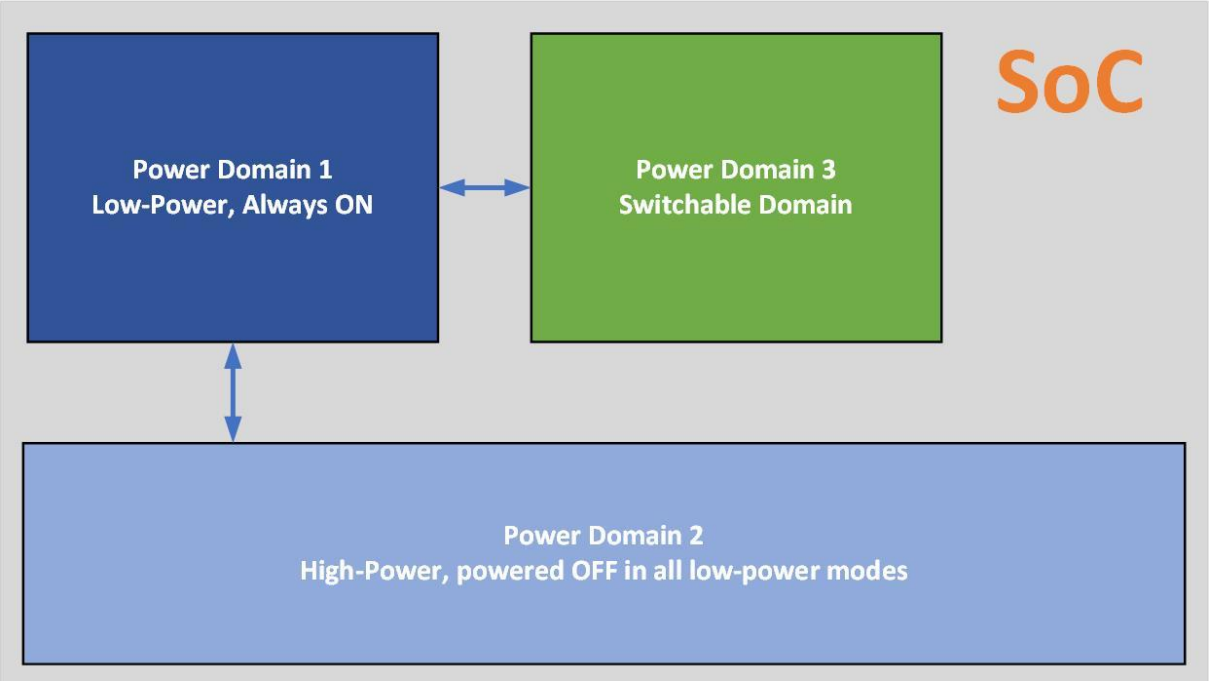

Figure 6. Typical power domain division in a system-on-chip.

2.3.4 Power managing operational modes

Low-power SoC designs usually have the chip, or sections of it, work on multiple operation modes, switching modes depending on the current operation. The switching is controlled by a Finite State Machine (FSM). The operating clock frequency is an important factor of switching power consumption, and thus the highest frequencies should only be used in the high power mode to avoid wasting energy. Furthermore, clocks can be disabled altogether for modules that are not needed in the current operation mode. A high-frequency clock can be generated by a Phase-Locked Loop (PLL) module, which multiplies a low-frequency clock to create a higher frequency clock. This can then be divided down to provide any needed clocks for modules. [19]

In addition to controlling the operating clocks, FSM modules in low-power designs may also oversee the control of power management structures. Power gating and used supply voltages are usually directly related to certain operational modes, and a switchable power domain may only activate during a single operational state.

Figure 7 shows a chart of state transitions in an example low-power SoC design. In this case, four modes are defined: SLOW, NORMAL, IDLE and SLEEP.

(1) SLOW: When the system finishes reset or does not need to run in a high clock frequency, the system enters SLOW mode. In this mode, the PLL module is disabled, and the CPU and other modules run at the slower frequency input clock.
(2) NORMAL: In NORMAL mode, PLL is enabled, providing the system with the maximum operational frequency. CPU and other modules run at this high frequency.
(3) IDLE: If CPU core finishes all tasks and would be idle for a long time, the system enters IDLE mode. The system can enter this mode from either SLOW or NORMAL mode. In IDLE, the clock to CPU core is disabled to reduce power dissipation. CPU core can be enabled with a reset or an interrupt, in which case the system returns to the mode prior to IDLE.
(4) SLEEP: If the whole system finishes all tasks and would be idle for a long time, the system enters SLEEP mode. In SLEEP mode, PLL is disabled, as well as all clocks in the system. SLEEP mode can be exited with a reset or an external interrupt and will switch the system to NORMAL or SLOW operation mode, depending on the current application. [20]

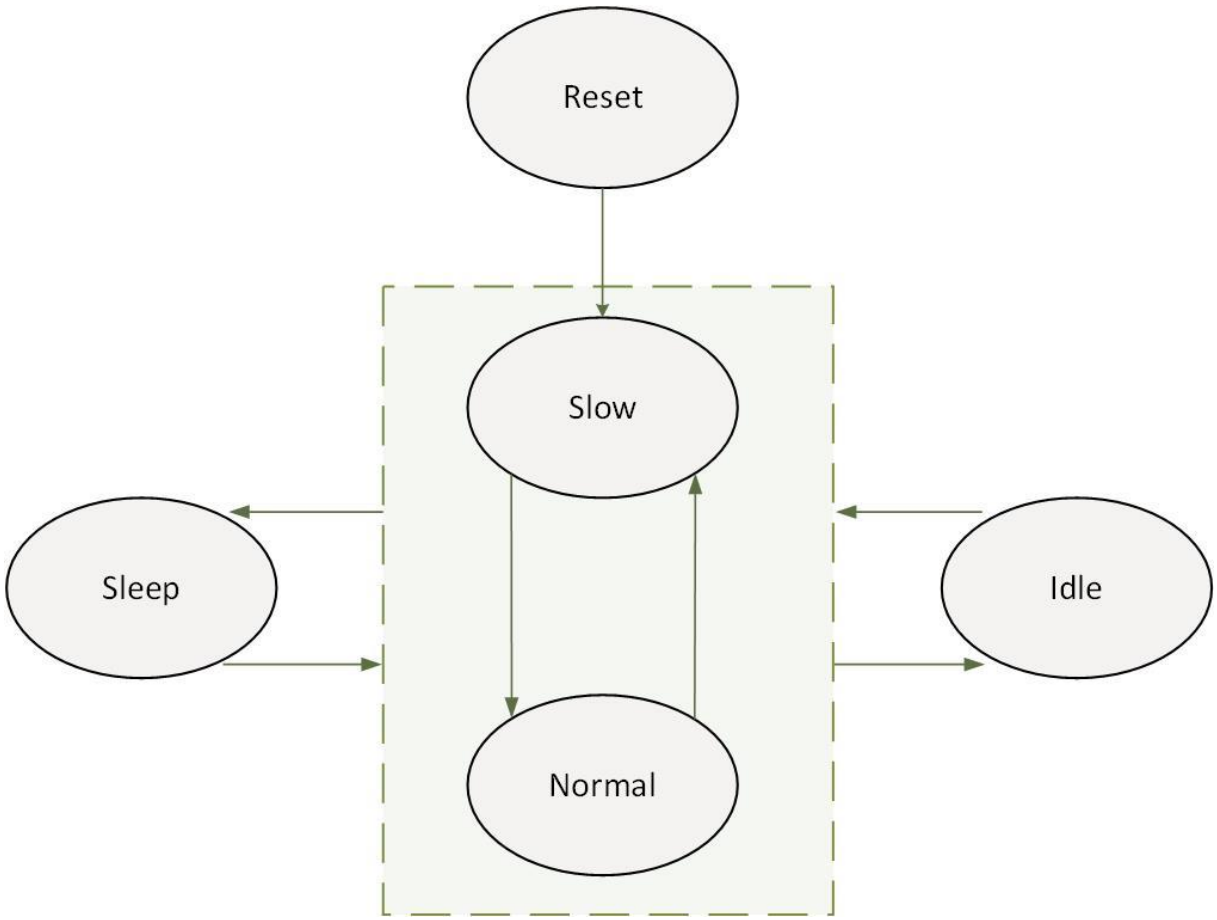

Figure 7: An example state transition chart for a low-power system.

3 SOC VERIFICATION

Chapter 3.1 provides an overview of SoC verification, and subheadings 3.1.1 and 3.1.2 introduce simulation-based verification methodology and its alternative, formal verification methodology. Chapter 3.2 shortly discusses verification challenges arriving from low-power methodology.

3.1 Introduction to functional verification

As the designs of SoCs have gotten highly complicated, it has become increasingly difficult to thoroughly verify the functionality of the chip and identify all the design bugs before the chip is sent to manufacturing. It has been estimated that in current industry designs, functional verification may take up to 80% or more of the overall design time [21]. Therefore, viable verification techniques for each stage of the design flow are indispensable, especially for earlier stages, since fixing design bugs in later design stages is very expensive.

IC verification is done on every major step of the design flow, from verifying the compilation of the first models on the Register Transfer Level (RTL), to the verification done on the manufactured physical chip before the final tape-out. Functional verification focuses on the design before any of its parts are built, attempting to determine if the design model operates as intended [22]. Most design and verification effort is done at register-transfer level, which presents the model on a less detailed abstraction level but detailed enough that all functional detail is usually available in the model [22]. The work in this thesis focuses on the early parts of the design and verification flow, thus only functional verification at the register-transfer level is explored. Verification methodologies can be generically defined either as simulation-based or formal methods [21].

3.1.1 Functional/Logic simulation

Simulation is an old and well-known but extremely useful verification method. The basic concept of simulation is quite straightforward and is presented in Figure 8. An electronic design automation (EDA) tool is used to create and exercise a simulation model of the design. A test case is created, where a set of inputs are given to the design under test (DUT) as input, and the resulting responses are captured and observed. These responses are then compared to a specification of the design to see if the DUT adheres to it. If the results differ from what was expected, there exists a bug in the design. After diagnosing the cause and editing the implementation to fix the bug, the simulation is run again. This process is repeated until the test case doesn't issue any bugs, after which this pattern is repeated for all simulation scenarios until finally the implemented design is bug-free. With enough different simulation scenarios, a certain level of confidence is gained for the correctness of the design. [23]

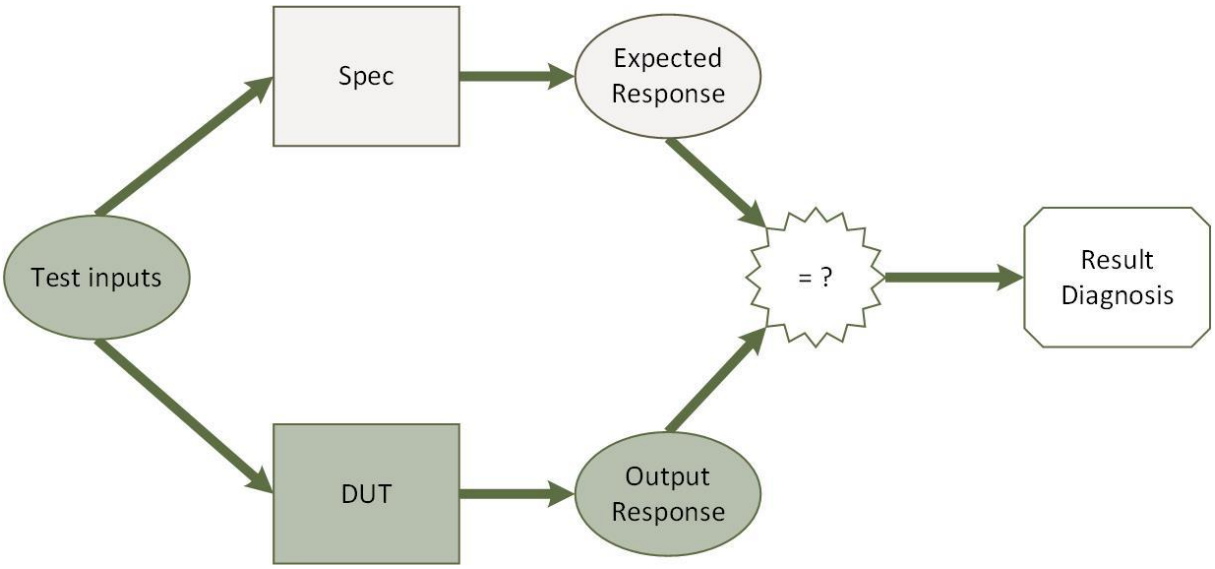

Figure 8: Basic concept of verification by simulation.

Although verification by simulation is a common and good practice, the ever-increasing complexity of SoC designs is starting to take its toll. Because the simulation results depend on given input patterns, the design might include bugs that are harder to find, or ones that, without a completely thorough verification, may even be completely invisible to logic simulation. The increasing SoC design complexity requires exponentially increasing input patterns, which makes it extremely difficult, or outright impossible to do thorough verification by logic simulation. The development of formal verification techniques is essential to solving this problem. [21]

3.1.2 Formal verification

Formal verification is a mathematical proof that two models, one that is implemented through the written code, and one that is created through design specifications, are identical under all conditions [24]. Formal methodology focuses on proving this using a mathematical process rather than a simulation-based approach where specific test cases are created and driven on a design.

In formal verification, the design and its specifications are translated into mathematical models, which are used to prove the correctness of the design with mathematical reasoning. As all possible cases in the mathematical model are explored, formal verification is basically exhaustive and can be considered as simulating all cases in logic simulation. [25]

Formal verification can be classified into two classes, model checking and equivalence checking. Model checking verifies if a design satisfies the properties given as its specifications, whereas equivalence checking verifies whether two given designs are equivalent to each other or not. The work in this thesis focuses on formal model checking, and thus equivalence checking shall not be discussed in any more detail. [21]

The formal model checking methodology uses properties, created from design specifications, to provide proof of design correctness. These properties are written as assertions, which are concise descriptions of complex or expected behaviour, and they can be used to present the functional intent of the design. These assertions can be static, meaning they must always hold true, or temporal, meaning they must hold true only at a specific instance and time.

A formal verification tool reads design properties in the form of these assertions and attempts to prove that they can never be violated. If such a case is found, the tool finds and presents a stimulus sequence that violated the assertion. This is termed as a "counter-example". If an assertion can neither be proved or falsified, the assertion is termed as "indeterminate". [25]

Figure 9 shows an example illustration of discussed behavioural checking. The design starts at an initial state and then walks through all possible actions and transitions one by one in an attempt to get the design to execute an illegal behaviour. If such behaviour cannot be executed by any means, the design can be viewed to be correct.

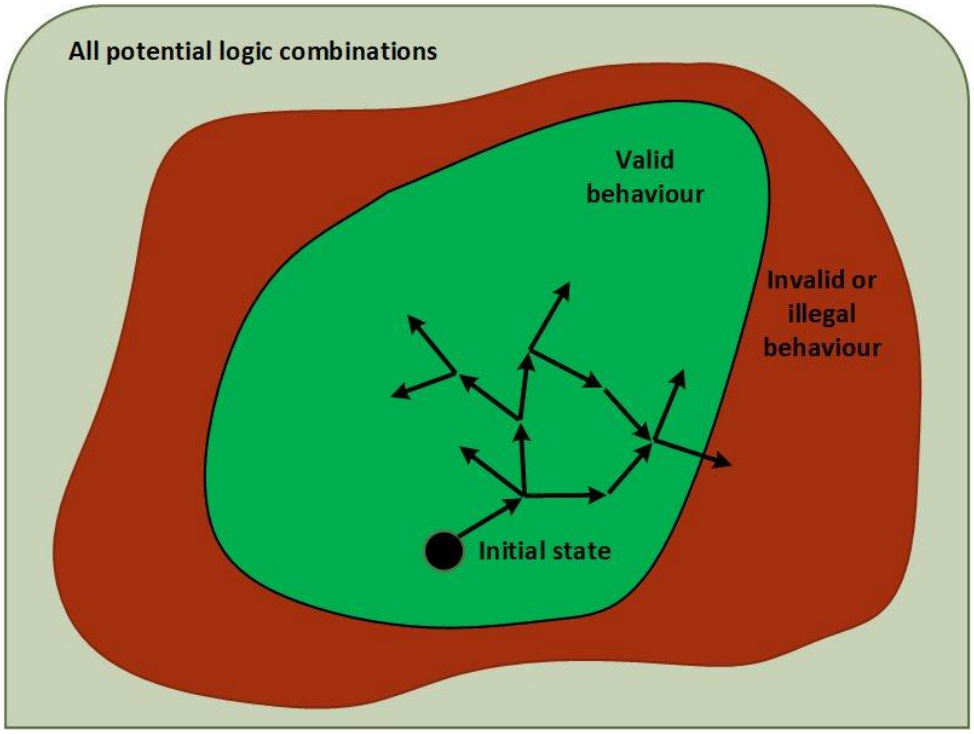

Figure 9: Formal Verification behavioural check illustration

3.2 Challenges of low-power SoC verification

The usage of power management features in low-power SoC designs brings up some unique challenges to verification. As the foremost issue, the design description languages at higher levels of abstraction don't have a notion of voltage as a variable. This is now being addressed through the development of power formats such as Unified Power Format (UPF) and Common Power Format (CPF), which reflect the power intent of a design. [26]

Besides the problem with description languages, low-power architecture itself significantly complicates verification activities. Low-power designs can feature tens of power domains and thus hundreds of power modes, making it prohibitive to verify that the design is functional under all possible power modes [27]. The verification of specific targets through logic simulation gets much more elaborate as well. In low-power designs, such as the one used in this thesis, multiple different power reduction techniques are combined and used in parallel, making the required configurations for verifying a specific target through simulation quite complicated. Even a seemingly simple target, such as verifying a clock in an IP, may require a lot of arrangements to be done before it can be tested and observed. The power domain under which the IP is operating must be enabled and its power turned on correctly. The clock signal to the

IP must be enabled correctly. The power domain must have the correct operation mode to enable the operation of the IP under verification. Finally, after all these configurations are set up, a test case can be created, where the IP receives a stimulus, and through its simulation results the activity of the clock signal can be observed. In addition, to ensure that the correct activity is preserved in the future, the testbench requires checkers to continuously check and verify the clock activity. As these checkers must operate under the specific condition configured for the operation of the IP, creating them can be difficult as well.

These types of configurations are needed to verify the connectivity of one clock signal in a specific IP through logic simulation, and they must be repeated for every separate IP in the design to achieve complete clock verification. Conducting this sort of connectivity verification on the entire SoC design can become a very difficult and time-consuming task without a suitable method. In addition, it can only be conducted fairly late in the design flow, when all the modules to be verified have fairly complete functionality.

4 FORMAL CONNECTIVITY VERIFICATION

Chapter 4.1 describes SoC integration and possible connectivity types inside the design. 4.2 introduces formal connectivity applications as a solution to verifying low-power SoC connectivity. 4.3 discusses security applications as a way to supplement connectivity checking.

4.1 Connectivity in SoC designs

One challenging part of SoC development is top-level integration and verification. There are multiple IPs which may have multiple owners. Each IP has input and output ports, which may vary from tens to hundreds. Incorrect specifications, misinterpretations of specifications, and misunderstandings between designers and verification engineers are just some of the reasons for design errors during integration. It's been observed in many in-house SoC designs that up to 80% of errors during the design integration process are contributed by pure connectivity errors [28].

Figure 10 presents possible types of pin to pin connections inside SoC designs. A connection is described by its source and destination nodes. Source and destination may be connected through flip-flops to create a delay, may have a conditional connection, or may be a combination of two or more of these types. [29]

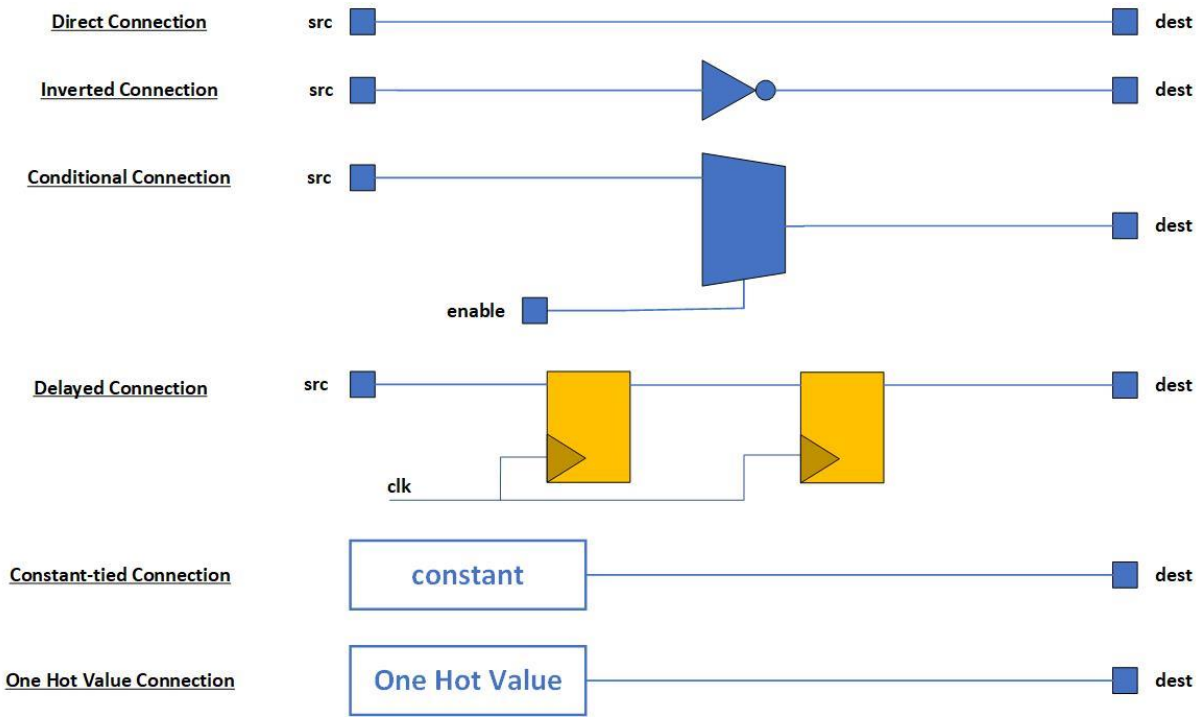

Figure 10: Possible types of pin to pin connections.

Verifying connectivity on the highest level of an SoC design can be very tedious. The verification engineer needs to have some insight into the entire design to even begin the task. The connectivity path may go through multiple blocks and levels of hierarchy. Inverters may exist along the path, and state elements like registers and flip-flops result in multi-cycle delays between the source and destination points. Global signals such as clock and reset are routed to thousands or millions of state elements, and the correctness of those connections should also be

verified. These reasons render connectivity verification by inspection completely impractical. [3]

4.2 Connectivity verification with formal applications

Today, EDA vendors of formal verification tools have started providing their software with specified “apps”, which target specific verification challenges. These apps typically generate most, or even all of the properties needed for formal analysis. This ease of accessibility allows even users with no prior formal experience to start using formal verification. One of these apps, connectivity checking, is one of the most widely used applications of formal technology. [3]

Connectivity checking, as the name implies, is an application targeted at ensuring proper interconnections among design blocks. In formal connectivity, a connection between a Source and a Destination is defined by three conditions:

- Source and Destination always have the same value, when a given Enable expression is true
- A structural path exists between Source and Destination
- The connection is directional from Source to Destination

If all these conditions hold, the connection can be deemed as proven, whereas if even one of these conditions fails, the connection is deemed as falsified. [30]

A formal verification tool uses given information on the Source, Destination and Enable expression, and automatically generates a property that exhaustively verifies the connection. The created property can be thought of as a SystemVerilog Assertion (SVA) of the following type:

assert property @(posedge clk) (enable |-> (source==destination))

where at every positive edge of the clock, if the enable expression is valid, source and destination should be equal.

The important difference is that the created connection property also indicates directionality and a structural connection, whereas with an assertion, the Source and Destination could as well both be tied to a constant. They would always have the same value, thus according to the assertion, they would have a connection. [30]

The design connectivity specifications themselves can be given to the tool in multiple ways, including Comma-Separated Values (CSV), Yet Another Multicolumn Layout (YAML) and Tool Command Language (TCL) formats. Spreadsheets in CSV format, as shown in Figure 11, have all property arguments separated into different columns, which brings a lot of ease to property writing and readability of the specifications.

	A	B	C	D	E	F
1	SOURCE	DESTINATION	ENABLE	NAME	DELAY	CLOCK
2	top.A.in1	top.B.out1	top.B.en	in1_to_out1	0	
3	top.A.in1	top.B.out2	{top.B.en, top.B.valid}	in1_to_out2	0	
4	top.A.in2	~top.B.out1	top.B.en	in2_to_out1	0	
5	top.A.in2	top.C.in1		in2_to_C_in	2	top.ck

Figure 11: An example of a CSV format spreadsheet

4.2.1 Connectivity property extraction

In an ideal situation, the connectivity that needs to be verified is defined in design specifications or separate connectivity specifications with perfect accuracy. The source and destination signals are defined accurately down to the correct bit-indexing, as well as all needed enable expressions on the path. Unfortunately, this isn't always the case, most likely during early parts of the design flow, where the design requirements are not yet solid, and the design structure and functionality are constantly being reworked. Due to the constant changes in the early design, the documentation can be very incomplete. For a verifying engineer who is not involved in the design process, finding out the correct connectivity specifications can be a very difficult and time-consuming task.

Synopsys' formal verification tool VC Formal™, which was used in this work, has a feature to automatically find and extract connectivity properties from actual connections present in the RTL. The tool takes a given source and destination in the form of signals or even whole module instances and tries to find any existing structural connections between them. If a structural connection is found, the tool creates a connectivity property from said connection, including any corresponding enable expressions. Naturally, the automatically created properties might not be consistent with specifications and must be reviewed to ensure they are correct. [31]

While results from properties that were created with accurate specifications are arguably more trustworthy, property extraction can save a lot of time during the early design stages, when design verification doesn't need to be completely thorough. It enables designers to focus on the design work and enables verification engineers to proceed with design verification without a need to investigate the design functionality and structure from written RTL-code.

4.3 Disproving illegal connectivity

When verifying proper connectivity in a design, proving only the existence of wanted connectivity may not be enough. For a complete connectivity verification, one may need to ensure that the selected source is connected *only* to the proper ports and nothing else. This means also proving the absence of incorrect connectivity. This type of check is non-trivial and may not be accomplishable by the connectivity application, forcing one to find other options. [32]

There are certain tools, or specific features of tools, that are focused on verifying data propagation in a secure manner, meaning they prove that data doesn't propagate between two points, where one is considered secure and the other non-secure [33]. Security verification tools of this type are exactly what is needed to prove that only the proper connection exists, and any other possible connections are absent in the design. These tools function in a similar way to connectivity checking applications, as they also create the properties to be checked from the given source and destination information [32]. The big difference is that instead of a wanted connectivity destination, they take in an unwanted or illegal connectivity destination, and verify that the connection doesn't exist. Combining this checker with connectivity checking gives an arguably solid proof for connectivity validity.

5 CLOCK AND RESET CONNECTIVITY VERIFICATION

Chapter 5.1 introduces the goal and requirements for the verification work done in this thesis. Chapter 5.2 describes the structure of the resource distribution system present in the used design. Chapter 5.3 presents the different stages that the verification work was done on. Chapter 5.4 discusses how the used connectivity properties were created. Chapter 5.5 provides an overview of the used process flow for the used software tool.

5.1 Verification goal and requirements

The goal in this thesis was to investigate formal connectivity verification as a viable method to verify clock and reset connectivity in an ultra-low-power SoC design, where power reduction solutions are pushed to their limits. To prove the viability of this verification method, the following goals needed to be reached:

1. Existence of wanted connectivity between a clock/reset source and its destination must be verified (proven or falsified).
2. Verification method must be usable on the highest level of the SoC design, where all parts/subsystems are integrated.
3. The procedure of verifying wanted connectivity must not take too much effort and manhours (even for a good method, too much effort can make it unbeneficial).

Complete proof for correct connectivity has an additional requirement where illegal connections must also be disproved, as presented in Chapter 4.3. However, due to restrictions on timing and software licenses, this requirement was left out from the work in this thesis and is discussed in Chapter 7 as a supplementary improvement in future research.

5.2 Structure of verification environment

The used verification environment for this thesis was an ultra-low-power SoC design, consisting of multiple subsystems. Each of these subsystems consists of multiple IPs, AMBA interconnect buses, CPUs, and their own power, clock and reset controlling systems. For ease of reading, from now on these signals shall be bundled together and referred simply as PRC-signals, after “Power, Reset and Clocks”. Each subsystem works as its own individual region and has its own power domain division for used peripherals. The distribution of PRC-signals for each subsystem is controlled by an outer resource management system, which grants these subsystems resources depending on their activity.

The PRC-signal distribution inside each subsystem follows a certain gating logic structure, which is described in a simplified form in Figure 12. Each power domain in each subsystem has a resource controller, with the main power domain having the controller with the “highest authority”. The main resource controller then handles incoming requests for PRC-signals from receivers, that can be peripherals, CPUs or separate power domains working “under” the main domain. According to the request protocol, the controller may distribute the requested resources to the receivers requesting them or deny them in the case of an invalid request.

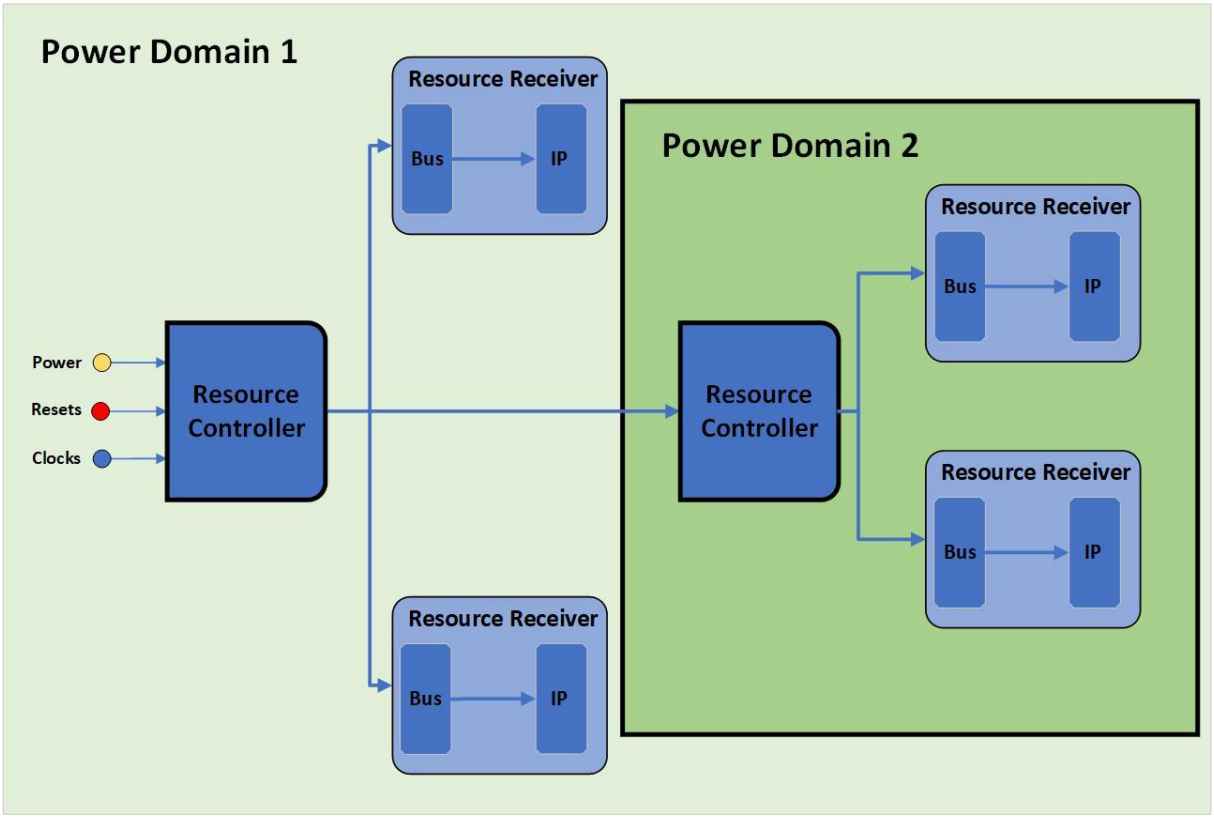

Figure 12: Resource distribution system in used verification environment.

The resource controllers, while controlling PRC-signal distribution, also control the operational mode of the domain they are in. By extension, they also control the power management structures of that domain. Each resource controller is individually parameterizable according to the needs of the domain it is placed in, making them much more complex and difficult to outline than how they initially seem.

While the subsystems have total control of their inner resource distribution, the subsystems themselves work "under" the main resource control unit, which activates and deactivates the subsystems according to their activity. This unit and its connections are illustrated in Figure 13. The activity of one subsystem may also have effects on other subsystems, such as asserting resets in the case of an error. These cross-domain requests and commands are directed to the main resource control unit, which then distributes resources to the subsystems according to the request or command.

5.3 Verification stages

The connectivity of clock and reset signals was verified on two different stages: first on the highest level of SoC architecture between the global resource distributor and each subsystem, and later within two separate subsystems. The environments in these two stages are illustrated in Figure 13.

The first stage was a simple and straightforward connectivity check to verify the correct integration and connections of subsystems on the topmost level of the SoC. Request signals from subsystems must be correctly connected to inputs of the global resource distributor. Likewise, the distributed clock and reset signals from the global distributor must be correctly connected to their corresponding inputs for each subsystem. With these requirements

successfully proven, the integration of subsystems on accord of clock and reset signals can be considered correct.

The second stage was focused on the correct integration of individual IPs within a subsystem. Each power domain within a subsystem has one resource controller module, and they are each individually configured according to the domain they are in and the IPs working under them. This makes it possible for a clock or reset signal path to go through multiple resource controllers, creating a lot of complexity to a seemingly simple connection. Each IP may also use multiple clocks and resets for different purposes, requiring verification for each of them.

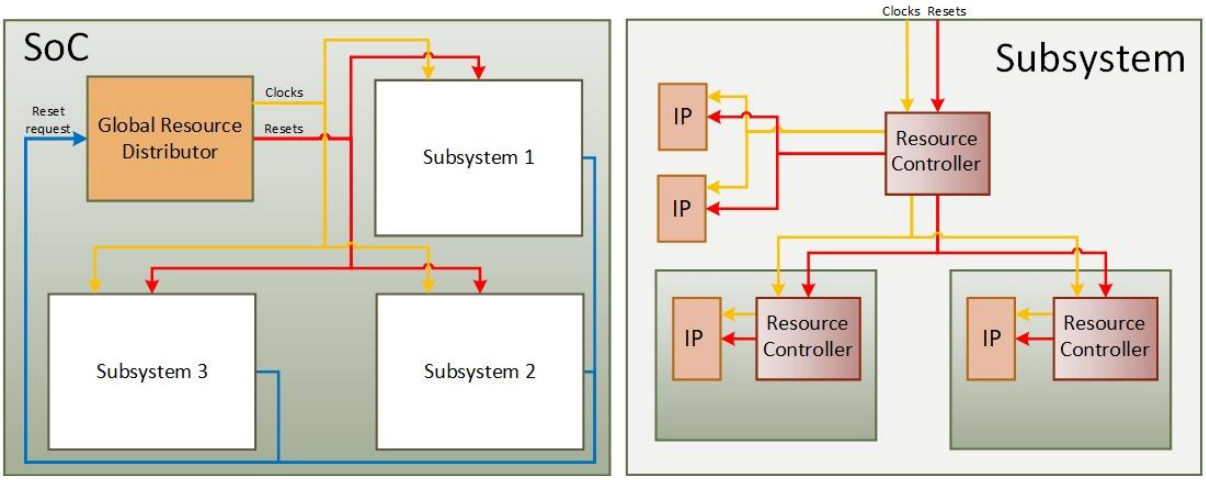

Figure 13: Verification environment on stage 1 (SoC) and stage 2 (Subsystem)

5.4 Creating connectivity properties

As explained previously in Chapter 4, connectivity properties can mainly be created in two ways: Writing accurate source, destination and enable expressions in CSV-format or some other compatible format, or using an extraction feature to automatically generate the properties and their enable statements. Both methods were used in this thesis, with each method being used on a different target.

Reset signals in the used design were asynchronous and didn't have special gating structures, therefore their connectivity properties didn't require any enable expressions or other conditions, such as latency. Reset-properties were created through a CSV-format spreadsheet, with accurate information on source and destination.

The property extraction method was used on clock signal properties. Although clock signals seem to have the same pathing as reset signals, unlike resets they have a strict and complex gating logic within each resource controller, requiring a huge combinational set of enable expressions. Clock signals also propagate through the design in an orderly signal package, each clock having a separate bit-wide part of the signal package. Their order in the package can also be scrambled inside each resource controller module, depending on the configurations done for the module. Writing accurate property-expressions, although possible to do, would be very tedious and time consuming for anyone not involved in the design process. By using property extraction, the right connection can be automatically found from all available signal-bits, including any needed enable conditions for the path, making the method very suitable for clock signal properties.

5.5 Formal verification tool process flow

As mentioned earlier in Chapter 4.2.1, the formal verification tool used in this thesis was VC Formal from Synopsys, more specifically its Connectivity Checking application. The used process flow for the tool was given in a single TCL-file, and that flow is presented in Figure 14. The process starts by defining the used application mode and all variables associated with it. Any variables affecting the software tool itself should also be set at this point.

Next step is to read and compile the design. The design is given as a set of all necessary files, or a single file list containing all the files. The tool will use given files and compile the design.

After compilation, clock and reset signals should be configured. Clocks are mainly used for properties that have some latency, and they are negligible in other scenarios. The targets in this thesis's design do not have any latency, so clock configuration was not necessary. Resets are used in the simulation phase to set the design to default stage before creating formal models. Resets in the used design were asynchronous, so they didn't require clock configuration either.

Next step is the simulation phase. During this phase, the design is simulated for a set amount of time to get the default state for formal analysis. By default, the previously configured resets are asserted during this phase, but any signal can be forcibly driven during this phase in order to get the design to a wanted state. The simulation can be run for a user-defined set of time, or until the design is deemed "stable".

After the simulation phase, the verification itself can begin. Properties can be loaded and created from a format of the user's choice. As previously stated, in this work CSV-format and VC Formal property extraction commands in TCL-format were used. After the properties are loaded, they are checked against the formal model created at the end of the simulation phase. Finally, the results of the check are presented and can be saved in an output file. Any properties that were successfully verified can also be used afterwards in a toggle-coverage analysis, the results of which are saved as a coverage database file.

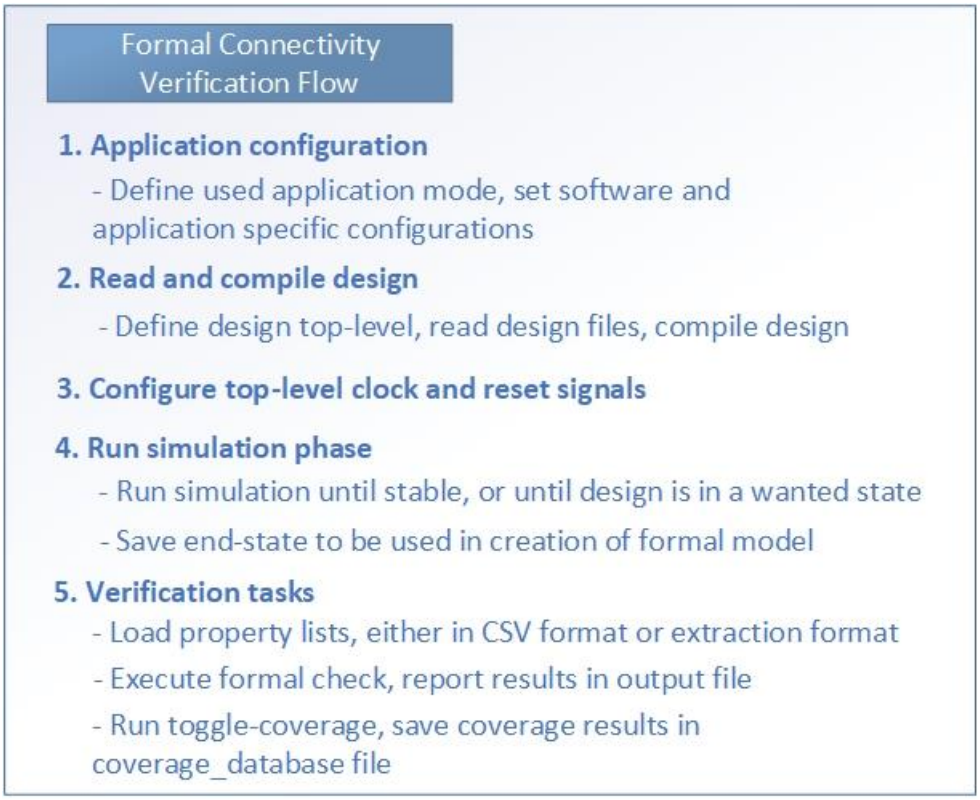

Figure 14: Used process flow for the verification tool

6 VERIFICATION RESULTS

Chapter 6.1 describes the procedures to confirm a falsified property as a design bug. Chapter 6.2 provides the concrete results of the task in this thesis. The number of verification targets and found design bugs for each verification stage is shown, and a short description of the details of the bugs. Chapter 6.3 compares the used verification method to the conventionally used in-house methods.

6.1 Bug confirmation

When the formal verification tool has completed checking the properties, it classifies them with either PROVEN, FALSIFIED or INCONCLUSIVE status. For any property that receives a FALSIFIED status, it must be confirmed to be due to actual design bugs, and not due to incorrectly written property or faulty specifications. The correctness of the property is easy to confirm, but the correctness of specifications is more difficult and is usually discovered only after reporting the discovered bug. In the situation where a fault in specifications is found, it is preferable to recheck all the properties, including the PROVEN ones, in case of false positives and false negatives.

After a falsified property is deemed as a design bug, the next step is to find the reason behind it. The formal verification tools produce a "counter-example" for the falsified property to support the debugging process [21]. The counter-example shows waveforms of the property signals, as well as any signals that had an effect with reaching the situation in the counter-example. By inspecting these signals, the cause of the bug can often be easily found.

Sometimes the counter-example is not clear at showing the cause, or it just isn't enough to decipher it. The formal tools may also feature a schematic viewer, with which one can directly investigate the signal path and even its values at every step. Some tools also feature automatic debugging features that can automatically search for the error cause. If none of these methods manage to find the origin of the bug, it should be consulted with the designer in charge of the DUT being verified. If the bug is due to a fault in the specifications, it would be found at least at this point of the debugging process.

6.2 Results of property checks

The gained verification results of the work in this thesis were divided into two, based on the two verification stages mentioned in Chapter 5.3. Table 1 shows the total number of verification targets for clocks, resets and reset requests on each verification stage, as well as the number in these targets that were deemed proven or falsified. Reset requests were present only at stage 1, and thus there are no targets for it included in the stage 2 section. The verification task resulted in finding two bugs in the used design, each on a different verification stage.

Table 1: Verification target amount on each stage, and the amount of proven/falsified targets

	Stage 1	**Stage 2**
Clocks	**9 (9/0)**	**80 (79/1)**
Resets	**18 (18/0)**	**62 (62/0)**
ResetRequests	**7 (6/1)**	**-**
Total:	**34 (33/1)**	**142 (141/1)**

The bug on stage 1 was found in the connectivity of reset request signals between one subsystem and the main resource controller module. During the integration of said subsystem, the signal was left unconnected, simply left to a floating state. However, by consulting with one of the designers of the subsystem, it was found to be in accordance with the design intentions, but the description of this signal connectivity in the design specifications was stated vaguely, or arguably incorrectly. This bug was thus deemed to not have been a bug at all and resulted from unneeded verification, resulting from conflicting design descriptions.

The bug on stage 2 appeared in the clock signal connectivity between a PRC-signal controller and a single IP working under it. The connection under verification was found to be missing, which was due to incomplete implementation of the IP under verification.

As a proof of concept, the used verification method has proven itself useful, being able to find design bugs even in a relatively complete design. Although both of these bugs can be considered as already-known beforehand, they serve well to give credibility to the used verification method.

6.3 Comparison to conventional simulation method

A comparison of the needed resourcing, tool runtimes and total work effort for the investigated method in comparison to the conventional simulation method is shown in Table 2. The information on the simulation side of the table was gathered from in-house verification and simulation statistics, as well as from consultations with colleagues in charge of design verification work. As seen from the comparison, the investigated formal method requires far less personnel and work effort to achieve connectivity validation. The conventional method requires a team of people to focus on the task, with each person having a different duty to fulfil. These duties may include creating the testing environment, creating test cases and stimulus for the DUT, and creating checkers and properties that survey the DUT for the possible design bugs. With the formal method, all of this can be achieved by just one person, who creates the formal environment and the connectivity properties for the DUT. Creating stimulus separately is not required, as the formal tool goes through all possible scenarios automatically to exhaustively prove or disprove the created properties.

Table 2: Work effort comparison between formal and simulation methods

	Formal verification	Simulation
Total personnel	1 person	Team (eg. 2-5 people)
Work task division	Formal env. creation and writing CC properties: 1 person	Stimulus/test cases: 1-3 persons Checkers/Assertions: 1-2 persons
Compile runtime	Subsystem: ≈5 minutes SoC/Top level: ≈30 minutes	Subsystem: ≈1 minute SoC/Top level: ≈5 minutes
Test case/Property-check runtime	Subsystem: <1 minute SoC/Top level: <5 minutes	Subsystem: ≈1 minute SoC/Top level: ≈30 minutes

Total time spent on verification task	With clear specs: 2-3 weeks Without clear specs: 2-4 weeks	Testbench/checkers: 2-3 weeks Testcases: N x Weeks

There are some big differences between formal and simulation tool runtimes, some in favour of the formal method and some in favour of the simulation method. Formal tools take a substantially longer time to compile the design, due to the creation of the mathematical models needed for the formal analysis. The runtimes for test cases and formal property checks are a bit more divided, where a smaller design is run faster through simulation, but the top SoC level is run faster through the formal method. At least on lower levels, such as on subsystem-level, simulation method seems to be faster.

However, two of the greatest advantages of the formal method can easily remain unnoticed in the result table. The first of these advantages is the number of needed test cases or property checks. Because the conventional simulation method doesn't verify connectivity directly, but through functionality verification, it requires the creation of a bunch of different test scenarios, which all need to be run separately. The formal method, however, can exhaustively prove *all* of the created properties in parallel and during only one run of the tool. This makes the formal method incredibly faster when there are a lot of verification targets, with the difference growing larger the more there are targets to be verified.

The total time spent on verification is also directly related to the number of needed test cases. Whereas the formal method requires the connectivity properties to only be written once for them to be exhaustively verified, the simulation method usually requires multiple test cases, all of which require work effort and time to be created. As the time spent with simulation method directly scales with the number of required test cases, the advantage of the formal method scales with the size and complexity of the used design.

The second advantage of the formal method is about its place and timing in the design flow. Because the simulation method verifies connectivity through functionality verification, it requires the modules under verification to be fairly complete before their connectivity can be verified. The formal method directly verifies connectivity, so it only needs the modules to be integrated to conduct verification. This means that the formal method can be used much earlier in the design flow to discover connectivity issues before they have a chance to negatively affect functionality verification.

7 DISCUSSION

The main goal of this thesis was to investigate formal connectivity verification in verifying clock and reset connectivity in an ultra-low-power SoC design. The work was done in order to find a better and easier alternative to conventional simulation-based methods for this kind of verification task. The conventionally used simulation methods during the verification flow do not focus on checking connectivity, but rather certify it as a side product of functional verification. As pure connectivity errors cause a huge percentage of errors during integration, a verification method focusing only on connectivity verification is highly welcome.

The gained results fulfil the aim and requirements set for this study. The method was proven to be usable even at the highest architectural level of the used SoC design, where all or most of the design components have been integrated. Pre-required experience of the design under verification and the verification technique itself was deemed quite low, providing ease of accessibility for new users. The resourcing needed for a task like this was shown to be substantially lower than through the conventional methods, only requiring a single person to conduct the task, rather than a team where each person has a different role to perform. The work effort and time requirements were effectively lowered as well, where the difference scales with the increasing size and complexity of the design. Although the formal method has limitations where it can't verify design functionality, it can easily check and discover basic connectivity issues that can't be exclusively sought with simulation methods. Also, perhaps most importantly, the formal method can be used at a much earlier stage of the design flow, where the bugs in connectivity are less harmful and easier to fix.

In a view of complete verification of clocks and resets in a design, this verification method would have its own suitable place in it. The complete verification of clocks and resets could roughly be divided into three different sections: clock or reset generation, their correct connectivity, and verification of their individual features (correct frequency and frequency monitoring for clock signals, reset cycles and correct system activity after a reset instance for reset signals). While the signal generation and the individual features would require functional verification, the investigated formal verification method easily fulfils the connectivity section of this verification plan, leaving the other two sections to be conducted through some other methods.

One valid criticism of the performed work would be the way that the property extraction feature was used in proving clock connectivity. Since property extraction creates the connectivity property automatically from a path that structurally exists in the design, it is not ensured that the found connectivity has the correct path or enable expression. Should it be possible, the created connectivity properties must be checked for whether they are consistent with the design. The used design in this thesis, however, did not have proper connectivity specifications to check the properties against, and thus their correctness was left somewhat vague. But as the goal of this thesis was to investigate formal connectivity verification more as a proof of concept, rather than as an already valid verification method, and as the goals and requirements were fulfilled, this point of criticism need not be further addressed.

For future use or research in this topic, the design to be used should have pre-existing connectivity specifications that would be used for property creation. At the very least they could be a part of the main design specifications, where a verifying engineer can see the connectivity descriptions in a table or described as text. The best scenario would have the design connectivity described as a list of input and output ports in a file form that can be directly used by the formal verification tool, such as a YAML or a CSV file. This would require some extra work from the

designers who would need to maintain this connectivity list, but it would save a lot of time and effort from the verification engineer in charge of connectivity verification.

As shortly mentioned in chapter 4.3, a connectivity checker alone doesn't provide complete connectivity validity and should be supplemented with a formal security verification tool. A security verification tool can be used to disprove unwanted or illegal connectivity, which is not a trivial task to do with a connectivity checking tool. This type of a tool was not used in the work of this thesis due to some previously stated restrictions, but it should be taken to account in any future research on this topic, or when making a verification plan for connectivity verification.

8 SUMMARY

This thesis contains a study of formal connectivity verification in ultra-low-power SoC designs. The aim was to investigate formal connectivity checking as a viable method for the verification of clock and reset signal connectivity, which lacks a consistent verification method in typical simulation-based methods. Requirements for verifying proper connectivity are presented, as well as requirements for proving the viability of the investigated method. The used verification stages and process flow are shown to fill these requirements.

An overview of basic SoC architecture is provided. The power consumption in CMOS circuits and its classification to dynamic and static power is explained. Due to the advancement of CMOS technology, the required supply voltage of a transistor is constantly being lowered. A lower supply voltage is compensated with a lower threshold voltage to maintain performance, which in turn exponentially increases static power consumption, creating a conflict between dynamic and static power. Common power reduction techniques are introduced and explained in detail.

Modern SoC designs have gotten very complex, resulting in the verification process becoming more difficult and taking a large percentage of total design time. The classification of functional verification into logic simulation and formal verification is presented, and the details of both are explained and compared. Low-power design methodologies have introduced problems to design verification, with some being fundamental problems in the description languages themselves, and others bringing difficulty to test case creation, even in seemingly simple scenarios such as connectivity verification.

Connectivity and integration in SoC designs are introduced. A large percentage of errors during integration is from incorrect connectivity. Connectivity comes in many different types and can be a combination of multiple of them. Issues with connectivity verification by traditional methods have given a rise to formal connectivity verification, which can verify connectivity through formal methodology with relative ease. The requirements for proper verification of connectivity are introduced, and how a formal tool fulfils these requirements.

A set of requirements for the practical work are presented, all of which must be fulfilled to show the viability of this verification method. Clock and reset signals were chosen as the verification targets, and the structure of the used design environment is presented. The task was conducted on two verification stages, first on the highest architectural SoC level, between subsystems and the main resource controller module, and second on subsystem level, between the subsystem inputs and modules inside the subsystem. The connectivity properties were created in two different ways, directly through CSV-file specifications for the reset signals, and through a property extraction feature for the clock signals. Finally, the used process flow for the chosen formal tool is presented and explained.

After conducting a property check, any falsified properties must be confirmed as design bugs. A process for this confirmation is presented with debugging features typical to formal verification tools. The results of the property verification in this thesis shows the investigated method to be valid as a proof of concept, as the done verification resulted in the finding of two design bugs. Although both found bugs could be considered as known beforehand, their discovery still gives credibility to the method. The comparison between the investigated method and the conventional simulation method shows the enormous differences in resourcing, tool runtimes and total work effort. As the formal method can verify connectivity directly rather than indirectly through functionality verification, it is incredibly faster, less resource consuming and less time demanding than the conventional method, and it can be used much earlier in the design flow, where connectivity issues are easier to fix.

9 REFERENCES

[1] Moore, G. (1965). Cramming more components onto integrated circuits. In: Electronics, vol.38, no. 8 (pp.114-117).

[2] Schlachter, Fred. (2013). No Moore's Law for batteries. Proceedings of the National Academy of Sciences of the United States of America. 110. 5273. 10.1073/pnas.1302988110.

[3] Anderson, T. (2019). Automated connectivity checking with formal verification.

[4] Rajsuman R. (2000). System-on-a-chip: Design and Test.

[5] Mutschler A. (Accessed October 1st, 2020) IP Subsystems: What Works, What Doesn't. URL: https://semiengineering.com/the-ip-subsystem-what-works-what-doesnt/

[6] Bennet P. (Accessed September 29th, 2020) The why, where and what of low-power SoC design. URL: https://www.eetimes.com/the-why-where-and-what-of-low-power-soc-design/#

[7] Päivänsäde V. (2016) Dynamic Power Estimation with a Hardware Emulation Acquired Switching Activity Model. University of Oulu.

[8] Piguet C. (2005) Low-Power CMOS Circuits: Technology, Logic Design and CAD Tools. CRC Press.

[9] F. Mitu, G. Brezeanu, G. Dilimot, L. Anghel and I. Enache, "Method to increase the switching speed of MOS transistors by dynamic bias of the bulk," 1995 International Semiconductor Conference. CAS '95 Proceedings, Sinaia, Romania, 1995, pp. 241-244, doi: 10.1109/SMICND.1995.494907.

[10] Sarwar A. (1997) CMOS Power Consumption and Cpd Calculation. Texas Instruments

[11] Keating, M. & Flynn, David & Aitken, Rob & Gibbons, A. & Shi, K.. (2007). Low power methodology manual: For system-on-chip design. 1-300. 10.1007/978-0-387-71819-4.

[12] T, Suguna & M, Janaki. (2018). Survey on Power Optimization Techniques for Low PowerVLSI Circuitsin Deep Submicron Technology. International Journal of VLSI Design & Communication Systems. 9. 01-15. 10.5121/vlsic.2018.9101.

[13] Jan Rabaey. 2009. Low Power Design Essentials (1st. ed.). Springer Publishing Company, Incorporated.

[14] Haataja M. (2016) Register-Transfer Level Power Estimation and Reduction Methodologies of Digital System-On-Chip Building Blocks. University of Oulu.

[15] A. R. Durgam and K. Choi, "Optimized clock gating cell for low power design in nanoscale CMOS technology," Fifth Asia Symposium on Quality Electronic Design (ASQED 2013), Penang, 2013, pp. 85-88, doi: 10.1109/ASQED.2013.6643569.

[16] Massoud Pedram and Jan M. Rabaey. 2002. Power Aware Design Methodologies. Kluwer Academic Publishers, USA.

[17] Kursun, V. and E. Friedman. "Multi-voltage CMOS Circuit Design." (2006).

[18] M. Lanuzza, P. Corsonello and S. Perri, "Fast and Wide Range Voltage Conversion in Multisupply Voltage Designs," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 2, pp. 388-391, Feb. 2015, doi: 10.1109/TVLSI.2014.2308400.

[19] Daying Sun, Shen Xu, Weifeng Sun, Shengli Lu and Longxing Shi, "Low power design for SoC with power management unit," 2011 9th IEEE International Conference on ASIC, Xiamen, 2011, pp. 719-722, doi: 10.1109/ASICON.2011.6157306

[20] Z. Yu and J. Wei, "Low power design and implementation for a SoC," 2008 9th International Conference on Solid-State and Integrated-Circuit Technology, Beijing, 2008, pp. 2184-2187, doi: 10.1109/ICSICT.2008.4735003..

[21] Pradhan, D., & Harris, I. (2009). *Practical Design Verification*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511626913

[22] Meyer, Andreas. *Principles of functional verification*. Elsevier, 2003

[23] Fujita, M., Ghosh, I., & Prasad, M. (2010). *Verification techniques for system-level design*. Morgan Kaufmann.

[24] Semiconductor Engineering (Accessed October 13th, 2020) Functional Verification. URL: https://semiengineering.com/knowledge_centers/eda-design/verification/functional-verification/

[25] Vasudevan, S.. (2006). Effective functional verification: Principles and processes. 1-256. 10.1007/0-387-32620-0.

[26] B. Kapoor, S. Hemmady, S. Verma, K. Roy and M. A. D'Abreu, "Impact of SoC power management techniques on verification and testing," 2009 10th International Symposium on Quality Electronic Design, San Jose, CA, 2009, pp. 692-695, doi: 10.1109/ISQED.2009.4810377.

[27] W. Chen, S. Ray, J. Bhadra, M. Abadir and L. Wang, "Challenges and Trends in Modern SoC Design Verification," in IEEE Design & Test, vol. 34, no. 5, pp. 7-22, Oct. 2017, doi: 10.1109/MDAT.2017.2735383.

[28] S. K. Roy, "Top Level SOC Interconnectivity Verification Using Formal Techniques," 2007 Eighth International Workshop on Microprocessor Test and Verification, Austin, TX, 2007, pp. 63-70, doi: 10.1109/MTV.2007.22.

[29] H. Saafan, M. W. El-Kharashi and A. Salem, "SoC connectivity specification extraction using incomplete RTL design: An approach for Formal connectivity Verification," 2016 11th International Design & Test Symposium (IDT), Hammamet, 2016, pp. 110-114, doi: 10.1109/IDT.2016.7843024.

[30] Nordstrom A. (2017) (Accessed October 13th, 2020) Are you formally connected? URL: https://www.techdesignforums.com/practice/technique/are-you-formally-connected/

[31] Synopsys Inc. “VC Formal User Verification Guide” Synopsys Inc. Mountain View, CA, 2020.

[32] Ikram, Shahid & Derrico, Joseph & Farhan, Yasmin & Ellis, Jim & Parikh, Tushar. (2019). Connectivity and Beyond.

[33] Synopsys (Accessed October 14th, 2020) VC Formal. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html